Productivity Engineering in the UNIX[1] Environment

Object Management in Local Distributed Systems

Technical Report

S. L. Graham
Principal Investigator

(415) 642-2059

"The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government."

Contract No. N00039-84-C-0089

August 7, 1984 - August 6, 1987

Arpa Order No. 4871

[1]UNIX is a trademark of AT&T Bell Laboratories

OBJECT MANAGEMENT IN LOCAL DISTRIBUTED SYSTEMS

Songnian Zhou, Roberto Zicari

Computer Systems Research Group
Computer Science Division, EECS
University of California, Berkeley (*)

ABSTRACT

Resource management is a central issue in operating systems design; it is even more critical in distributed systems, because of the physical distribution of the resources, and thus the natural redundancy and the possibility of partial failures. In this paper, we study the resource management and sharing problems in distributed systems. After setting up a model for resource management that enables us to study and compare the different approaches, we survey the existing distributed systems and attempt to taxonomize the research results so far. Based on this investigation, we propose a new approach to resource management, which we call *global object management*. A logically centralized system-wide manager acts as a coordinator between different parts of the system, and is responsible for managing the top level, sharable resources in the distributed environment and making them available to the users. We argue that this approach greatly enhances system resource sharing by combining the semantic simplicity of centralized management and the reliability and availability of distributed management, and offers a number of advantages over the existing techniques.

1. INTRODUCTION

The past two decades saw a proliferation of computing resources as the cost of hardware dropped. The mode of computing is changing from a centralized structure to a distributed one. As a result, resources and services that are unaffordable by a single host may be shared by a group of hosts and provide satisfactory services to all the users. This poses new and difficult problems in the management of system resources that were not present in centralized systems. Because of the physical separation of the hosts, the global state of a distributed system, that is, the collection of states of the sharable resources in the system, is very difficult, or even impossible to maintain at a single location. Consequently, management decisions have to be made based on

(*) This work was partially sponsored by the University of California, Berkeley (Micro grant), C.N.R., Italy, and the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4031, monitored by the Naval Electronics Systems Command under contract No. N00039-82-C-0235. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Government.

incomplete and inaccurate information. Even resource location may be a difficult problem, because the resources are scattered around and may be mobile. Heterogeneity adds another dimension to the difficulties. Finally, partial failures affect the operations of other parts of the system and, even worse, inconsistency may occur as a result. In this paper, we study resource management problems in distributed systems. After setting up a model and conducting a survey of the existing techniques, we describe a new management approach that we believe offers a number of advantages over the present ones.

Before launching into a study of resource management problems, however, it is necessary to specify the type of systems in which we are interested. We define a *local distributed system* (LDS) to be a computing environment shared by a community of users and consisting of a multitude of hosts and devices connected by one or more local area networks (a local internet). In particular, we are not interested in multiprocessor systems characterized by shared memories, or computers connected by wide-area networks because such networks are usually slow, thus making it impractical to have a high level of resource sharing. LDS's are becoming increasingly popular, as computers of various types are purchased by an organization (for example, a university or a company) to satisfy its computing needs, and as the desire arises to integrate them into a distributed environment reflecting its organizational structure. Several characteristics of LDS's are worth noting:

- *Inter-host communication is usually fast and efficient.* This is true with high performance local networks, such as the Ethernet. As more hosts join the system, a single LAN becomes inadequate and several LANs have to be connected by a local internet via gateways. Although these communication components are significantly slower than LANs, technologies are becoming available to make them compatible with LANs in performance. Efficient inter-host communication is vital for resource sharing; if the communication overhead is high, the incentive for performing operations remotely quickly diminishes.
- *The scale of LDS's is large but naturally limited.* Since computer systems should complement the structure of the organizations they are supporting, and since, as a human organization grows over a threshold, it is usually divided into a number of suborganizations to ease the management task, LDS's have limited size. Strong locality of communication and resource sharing in distributed systems is evident. Together with the performance penalties imposed by the probable geographical separations, the organizational autonomy and the differences in work interests sharply reduce the communication requirements between organizations, and make interorganizational resource sharing less desirable.
- *The hardware and software environment is most likely heterogeneous.* Different parts of the organization supported by an LDS often have different needs and requirements that can only be satisfied by different types of equipment offered by multiple vendors. It is next to impossible to find a single vendor capable of meeting all computing needs. For example, a large number of workstations of varying capabilities are needed, as well as large hosts acting as servers. Heterogeneity makes resource management and sharing more complicated.

Besides the above system characteristics, several properties of an LDS are desirable.

- *High levels of resource sharing are crucial in an LDS.* Although hardware is becoming cheaper and cheaper, it is far from being free. Moreover, applications are emerging that

require more and more system resources. Most noticibly among these applications are artificial intelligence, image processing, and computer aided design. Just like in any other field, computer resources are going to be relatively scarce for a long time to come, and hence sharing the available resources is an important way to achieve economy of scale. More importantly, perhaps, the value of some types of resources is in sharing. Examples of such resources include files, databases, and software packages.

- *The system should be easily extensible.* As a human organization grows, and thus more resources are needed, new hosts and devices have to be added to the system. This should not cause disruptions to the existing system, while the new resources should easily become an integral part of the system.
- *A high level of system transparency, as well as an adjustable level of local autonomy, is desirable.* Transparency hides the complexity of the distributed system from the user, while providing all the resources. On the other hand, the resources often belong to different groups of users within the organization, so people want to retain a certain degree of autonomy. This means that each host should be able to specify to what degree it is willing to share resources with others. We call this *conditional transparency.*

In the next section, we present a model of resource management that will provide a framework for our studies. We conduct a selective survey of distributed systems design in Section Three, with an emphasis on resource management. We also summarize our findings in the survey and come up with a taxonomy. The global object management approach is presented in Section Four. Finally, we review the main results of the paper and future research directions.

2. A MODEL OF RESOURCE MANAGEMENT

In the previous section, we have identified the class of distributed systems for which we consider the problem of resource management. We have informally introduced the concepts of *sharable* resources and of *global state* of a distributed system. In this section, we want to develop a more formal framework for discussing the issue of resource management. We define resources as data abstractions (i.e., algebras) and we model a computer system as a hierarchy of such data abstractions. Furthermore, we give some basic definitions that will be used in the rest of the paper.

The section is organized as follows: in Subsection 2.1, we introduce the basic notion of abstractions and related concepts. In Subsection 2.2, we introduce the elements of the model, giving for each a formal definition. In Subsection 2.3, we define some concepts that will be used later in the next sections to characterize the resource management schemes of existing distributed systems.

2.1. Data Abstractions

It is fairly common to model computer resources as abstract data types. An abstract data type is composed of a set of values and a set of operations applicable to the values. The *definition* of an abstract data type consists of a *specification* of the type and an *implementation* of the type. The specification is visible to the user and gives the syntax and semantics of a set of primitive operations applicable to the type. The implementation is hidden from the user, and

consists of the implementation of the type's primitive operations on a selected representation for the set of objects. Several specification methods and languages for specifying data abstractions have been developed /A1//A4/. When an abstract data type is specified algebraically, it is viewed as a data abstraction, i.e., an algebra /A1/.

A data abstraction A is an algebra, that is, a set of possible legal states and operations to manipulate them. More formally we define a data abstraction A as a pair $<\overline{A},Oa>$, where $\overline{A}$ is a set of values called *states,* and Oa is a set of functions called *operations*. Generally speaking, an operation, applied to a state with some parameters, yields as a result a (possible) new state of the data abstraction, and some output values.(*) (**)

In order to characterize some basic properties of data abstractions, let us introduce some definitions. We will refer to the definitions given in /A2/.

DEFINITION 2.1 Given two data abstractions A and B, A *statically includes* B through α , iff there exists a surjective function α from A to B. The function α is called *abstraction function.*

The notion of static inclusion is a necessary condition for having a data abstraction A as a possible representation for another data abstraction B. It is a static property and only concerns the mapping between states of a data abstraction.

DEFINITION 2.2 An *expression* E on A is a finite series of operations oi:
$E = [o1,o2,...,on]$ where $oi \in Oa$, for $1 \leq i \leq n$.

DEFINITION 2.3 A *translation function* τ from A to B is a function that maps operations of B into expressions of A.

The abstraction function and the translation function are the basic notions to define the concept of *implementation* of a data abstraction.

DEFINITION 2.4 An abstraction $A=<\overline{A},Oa>$ *implements* an abstraction $B=<\overline{B},Ob>$ through α and τ, if and only if α is surjective and, for each $o \in Ob$, for each $b \in \overline{B}$, and for each $a \in \overline{A}$, $b=\alpha(a) \rightarrow o(b)= \alpha(\tau(o(a)))$.

That is, for any operation o of B, for any state of B for which the operation is defined, for any state of A, if the state of B is an abstraction of a state of A, then the operation on B produces a state, which corresponds to the abstraction of the state produced by the translation of the operation, applied to the starting state of A. α is a homomorphism with respect to τ, and the diagram of Figure 2.1 commutes.

We give now the notion of *equivalence* of data abstractions. We introduce two concepts of equivalence, static and dynamic.

DEFINITION 2.5 An abstraction A is *statically equivalent* to an abstraction B iff the function α is bijective.

(*) In the rest of the paper, we will use the convention defined in /A2/, where the name of an operation represents the pair 'operation generator, parameter', thus allowing us not to consider explicitly input parameters when dealing with operations.

(**)In the algebraic approach, states can be represented as equivalence classes of expressions which are specified by equations. In this paper, however, for simplicity, we assume that states and operations are defined extensionally. This simplification does not affect the generality of the definitions.

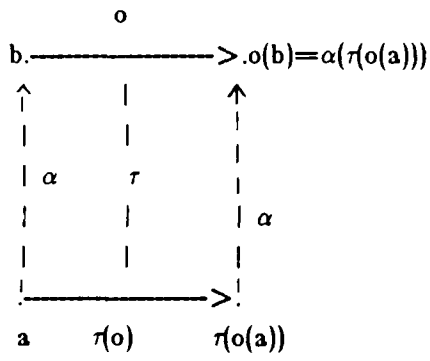


Figure 2.1

In order to express the concept of equivalence of two data abstractions including the operations, we need two translation functions, $\tau 1$ from A to B, and $\tau 2$ from B to A.

DEFINITION 2.6 Two data abstractions A and B are *operationally equivalent* if A implements B through α and $\tau 1$, and B implements A through $1/\alpha$ and $\tau 2$.

2.2. Elements of the model

In this subsection, we introduce a basic model of resource management. The elements of the model are illustrated in Figure 2.2.

(user)—>Manager—>Resource—>Handler

Figure 2.2

We define formally each element together with its interface.

A computer resource R, of any type, is modeled as a data abstraction $<\overline{R},Or>$, where the members of the set $\overline{R}$ are called the resource *states* and the members of the set Or are called the resource *operations*. The set Or represents the only operations allowable on the resource R. For example, for a sequential file, the set Or is indicated in Figure 2.4

An instance of a resource type has associated a unique *identifier (or name)* n(R). A resource can be referred to by name (*).

A resource is usually implemented by more primitive objects and their operations. For instance, a file is implemented in terms of disk blocks and the operations applicable to them. In the model, the *implementation* of a resource R is defined as the implementation of the data abstraction $<\overline{R},Or>$ which represents the resource, on another data abstraction $<\overline{H},Oh>$ which models another resource , wich we call the *handler*.

(*) Names can be complex /D4, D14/; we do not consider here the problem of the structure of names.

In general, the implementation of a resource is an iterative process. In fact, the resources in a computer system are naturally organized in a hierarchical structure. At the lowest level are the primitive physical resources such as bits of memory, devices, and computing elements. Because these resources are difficult to use, logical resources are built on top of them with a different level of abstraction. Logical resources are in turn the basis for the implementation of higher level logical resources. The levels of the hierarchy represent the degrees of abstraction of the resource types. It is useful to notice that, for the purpose of the model, hardware, firmware, and software implementations are all logically equivalent. The resource-handler scheme illustrated in Figure 2.2 is in reality a complex one.

Resources at the same level of abstraction with identical properties are often grouped together. The concepts of resource equivalence and of grouping can be formalized as follows.

DEFINITION 2.10 Given two resources R and H, R is *equivalent* to H, denoted by $R \equiv H$, iff R is operationally equivalent to H.

Given two resources $R=<\overline{R},Or>$ and $V=<\overline{V},Ov>$ implemented on the same handler $H=<\overline{H},Oh>$, we have the following property.

THEOREM 2.1

$R \equiv H$ and $V \equiv H \rightarrow R \equiv V$.

The proof of the theorem is omitted, since it can be found, in a more general case, in /A5/.

A resource R may be implemented on more than one handler; a typical case is when the system is heterogeneous.

DEFINITION 2.11 Given a data abstraction B, and n data abstractions Ai with $1 \leq i \leq n$, B is implemented on Ai, if there exist n surjective functions αi, with $1 \leq i \leq n$, and n functions $\pi i : Ai \rightarrow B$, for $1 \leq i \leq n$, and for each $o \in Ob$, for each $b \in \overline{B}$, and for each $ai \in \overline{Ai}$,
$b=\alpha 1(a1)=...=\alpha n(an) \rightarrow o(b) = \alpha 1(\pi 1(o(a1))) = ...= \alpha n(\pi n(o(an)))$.

When two resources R and V are implemented on more than one handler, Hi, $1 \leq i \leq n$, the following theorem holds.

THEOREM 2.2

$R \equiv H1 \equiv ... \equiv Hn$ and $V \equiv H1 \equiv ... \equiv Hn \rightarrow R \equiv V$.

So far we have characterized the process of abstraction as the basic method for building a hierarchical structure of resources.

We want to model now how a resource is controlled (independent of its internal structure). Generally speaking, the control of a resource is performed by another entity, that in the model we call *manager*. A manager controls and allocates the resource(s) for the user. An abstract manager has some interesting properties:

(1) it provides a set of operations for accessing the resource which are (possibly) different from the set of operations defined for the resource;

(2) it provides a (possibly) different structure of the resource;

(3) it performs some functions for controlling the resource (e.g. access control, allocation, scheduling, and so on) which are based on the state of the resource itself (and possibly of

other resources in the system).

From the first two properties, it is clear that a manager can be modeled as a data abstraction implemented on top of the resource it controls. Its function is to provide a different "view" of the resource. Furthermore it is also clear from the last property that a manager is something different from a pure data abstraction, as it acts as a controller and decision maker. An example of a hierarchy consisting of two managers ("Directory Service" and "File Server"), a resource("sequential File"), is illustrated in Figure 2.3. , and the corresponding operations are illustrated in Figure 2.4.

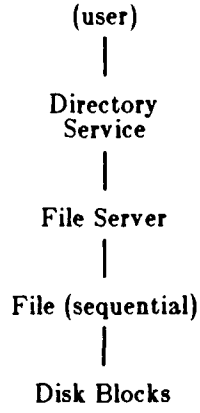


Figure 2.3

File Server Operations:

Create_file(ret FID)
Read_file(FID, ret data)
Write_file(FID, data)
Delete_file(FID)
Open_file(FID)
Close_file(FID)
Begin_transaction(ret TID)
End_transaction(TID)
Abort_transaction(TID)

Sequential File Operations:

Add(FID, data)
Remove(FID, data)
First(FID, ret data)

Figure 2.4

Managers can be modeled as data abstractions. A manager is identified by a unique name and can be referred by it. The concept of equivalence previously defined apply to managers too.

However, managers, though modeled as data abstractions, have some fundamental differences with respect to resources:

(a) They are *active* objects, in contrast to resources, which are *passive* objects. In the rest of the paper, when we refer to both active and passive objects, we use the generic term "object". The term "active" means that the object can change both its state and the state of other objects. A passive object cannot change state on its own, but only through an active object.

(b) They perform invocation, control, and allocation of resources on the basis of information about the state of the resources. (A more precise definition of the state of a collection of objects will be given in the next subsection).

For this reason, we extend the model to capture the characteristics of resource managers. As we have pointed out, a manager performs operations on the basis of the information about its state and the state of other objects in the system. We can formalize this concept, introducing the

notion of *conditional* operations. A conditional operation is executed or not depending on the result of the evaluation of a *predicate* on the object state.

If $A=<\overline{A},Oa>$ is a data abstraction representing an object, then Pa denotes the set of decidable predicates over A. Essentially, Pa consists of all predicates p that allow one to distinguish a particular subset of states $S\subseteq \overline{A}$, such that, for all elements $s\in S$, p(s) is true, but, for all elements $s'\in \overline{A}-S$, p(s') is false.

Thus, for *active* objects, we extend the set of expressions E (def. 2.2) to the set $E+ \supset E$, such that E+ contains all possible conditional operations that can be built from the operations of E (*). Given two operations o1 and o2, we define a conditional operation p[o1,o2], where p is a predicate, with the following semantics:
p[o1,o2]: for each $a\in \overline{A}$: p[o1,o2](a) = if p(a) then o1(a) else o2(a).

A *conditional expression* is a sequence of operations which contains at least one operation of the form p[o1,o2].

We will not deal with modeling the different schemes of communication and invocation between objects /D5/. Instead, in the next subsection, we concentrate on some other aspects of the resource management problem, namely, the process of grouping objects, their control, the definition of the state of an aggregate of related objects, and the object location, in order to capture the distributed nature of the systems we consider.

2.3. Object Control and Association

In order to refer to an object, its name must be known.

DEFINITION 2.12 A name space or *domain* D is a set whose elements are object names (**).

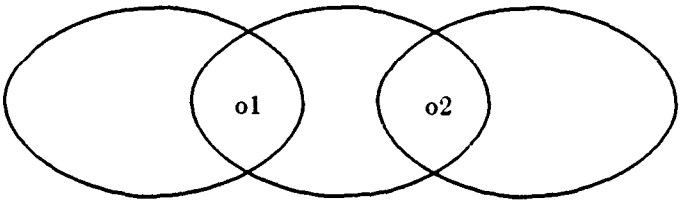
An object can be referred to by another object only if its name is present in the domain of the caller object. Often a domain is constructed from other domains.

DEFINITION 2.13 A domain D2 is *derived* from a domain D1, denoted by $D1\rightarrow D2$, if there exists a surjective function θ: $D2\rightarrow D1$ such that, for each $d\in D2$, there exists $d'\in D1$ such that $\theta(d)$ is defined and $\theta(d)=d'$.

A derived domain can be used, for instance, to model aliasing of names. An object name may belong to more than one domain.

DEFINITION 2.14 An object Oi is *sharable* with respect to the domain Di, with $1< i \leq n$, denoted by Ois{Di}, if its name belongs to the set of domains Di. An object is not sharable with respect to the domains Di, with $1 \leq i \leq n$, if its name does not belong to Di. In Figure 2.5, o1 is a sharable object with respect to the domains d1 and d2, and not sharable with respect to the domain d3, while o2 is sharable with respect to d2 and d3, but not with respect to d1. The *diffusion* of an object is the set of domains in which its name is present.

(*) The extension of the translation of the set E+ is not discussed here for reasons of space, but is described in detail in /A5/.
(**) The definition of domain differs from the one given in /D5/, where a domain represents the collection of states through which an active object evolves during its lifetime



d1 d2 d3

Figure 2.5

Let us now characterize the relationships between different elements of the model. In particular, we want to express the way an object's control takes place.

DEFINITION 2.15 A *group* G is a set of objects which are operationally equivalent.

DEFINITION 2.16 An *association* A is a mapping between groups of objects belonging to the same domain : A: G1 → G2. The mapping can be 1:1,1:n,m:1,m:n, corresponding to a variety of schemes.

An association is *oriented* when one of the groups is composed of active objects. The orientation of an association identifies the flow of control for objects. In order to make this point clear, let us consider Figure 2.6a, where two groups G1, composed of an active object a1, and G2 composed of three passive objects p1, p2, p3 are illustrated. The two groups are related by the association "a". The association "a" is oriented from G1 to G2. The meaning is that some of the objects belonging to G1 (in this case only a1) can control (and may alter the state of) some of the objects of G2, but not vice versa. In the example, the orientation of the association is the only possible one, since passive objects cannot modify the state of other objects. In the case of two groups of active objects related by an association, as in Figure 2.6b, the orientation of "a" expresses a constraint on object control: a1 and/or a2 can control some of the objects a3, a4, a5 but not vice versa. A bidirectional association, as in Figure 2.6c, means that some objects of group G1 can alter the state of some objects of group G2, and vice versa some objects of G2 can alter the state of some of the objects of G1.

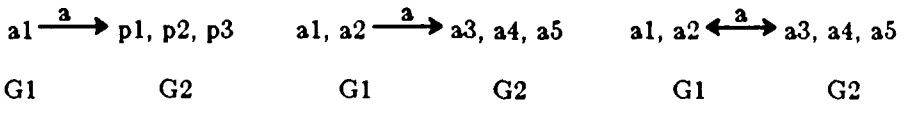


Figure 2.6a

Figure 2.6b

Figure 2.6c

The orientation of an association takes into consideration only object control and does not express the (possible) interaction between groups via output values. That is, we do not take into account the output values that a controlled object may produce and send to the controlling object. Note also that associations between groups of only passive objects do not have orientation.

DEFINITION 2.17 A *structure* H is a set of groups related by associations.

In our study, we are particularly interested in structures composed of groups of passive objects (resources) and groups of active objects (managers).

We now define what we mean by the concept of state of a "collection" of objects.

DEFINITION 2.18 The *state of a group* is the union of the states of the objects in the group.

DEFINITION 2.19 The *state of a structure,* S(H), is the state of the groups composing the structure.

In order to capture the distributed nature of a system, we must be able to define the real "location" of an object. The concept of domain, in fact, concerns only names and is independent of the location of an object.

DEFINITION 2.20 An *environment* e is a set of objects.

An environment can be associated with each host in a distributed system, and consists of all the objects physically located on the host.

In order to identify an object's location given its name, we introduce a function ξ called the *placement*.

DEFINITION 2.21 A *placement* ξ is an operation that, given as input parameter an object name, produces, as a output, an identifier of an environment.

When every host has an associated environment to it, the placement operation, given an object name, identifies the host on which the object resides.

DEFINITION 2.22 An object is *replicated* if the placement operation applied to its name yields as a result more than one environment. ξ (name) $\rightarrow$ {e1,...,en} where the ei's are environments.

Other operations can be defined to characterize the distributed nature of the system. For instance, the static or dynamic allocation of objects to hosts, the possibility of moving objects from one environment to another ,and so on. All these can be easily represented in our model as performed by managers, thus defining the manager's operations, and depend of the particular implementation. We will not describe them further, but it should be clear that the model, when applied to a specific system, will specify for each object (active or passive) the precise set of operations that can be performed on that object.

In this section we have introduced some basic concepts to define a formal framework that will be used in the next sections to characterize the resource management schemes of some existing systems. A more detailed description of the model is outside the scope of this paper, and is the subject of an ongoing research /D4/.

3. OVERVIEW AND TAXONOMY

This section presents a study of the resource management schemes in distributed systems. A brief overview of a selected group of systems is given in Subsection 3.1. Some of these systems (LOCUS, Eden, Cambridge, V System, Berkeley UNIX (*), and Xcode) are used in the rest of the section, which is organized around a taxonomy of the resource management schemes, with the goal to show both differences and similarities among the various solutions. A reader interested in more detailed descriptions of the systems should refer to the original publications.

3.1. An Overview of Distributed Systems

This overview includes, besides the systems listed above, additional systems that provide interesting resource management schemes relevant to this study. However, we do not claim that all the relevant systems described in the open literature are covered.

LOCUS

LOCUS is a distributed operating system designed and implemented at UCLA. The first system specification is dated 1981 /L1/, and a complete description of its architecture was published in 1983 /L3/. The goals in designing LOCUS were essentially the following:

- to make the development of distributed programming the same as that in a single machine environment;
- to provide high reliability and availability in a distributed environment.

The system architecture is based on a distributed file system, and provides application code compatibility with the Berkeley UNIX system. The major type of resource in the system is the file; other resources, such as IPC sockets and remote devices, are treated in the same way as files. LOCUS offers a set of non-standard functionalities:

- *network transparency:* The system provides a uniform interface to all resources. Access to a resource is independent of its location; the network is invisible at the application level. The resources have globally unique names in a single hierarchical name space. The component of the system which deals with network transparency is the distributed file system.
- *recovery and reliability:* The system provides automatic replication of files, including a global synchronization policy. Partitioned operations and mechanisms for reconciling inconsistencies after merging are also supported.

The system is operational at UCLA on a homogeneous network of VAX/750's (*) connected by an Ethernet. Further development of the system is currently being done in an industrial environment as well as at UCLA.

The resource management scheme that we will analyze in LOCUS is that of file management. The global naming hierarchy is partitioned into subtrees called file groups, each of which is managed by a centralized agent called a Current Synchronization Site (CSS). The CSS of a file group is in charge of file replication, consistency, and update synchronization. The locations of the CSS's are stored in the mount table that is fully replicated on every host.

(*) UNIX is a trademark of AT&T Bell Laboratories.
(*) VAX is a trademark of Digital Equipment Corporation.

Eden

The Eden system is an object-based distributed system designed and implemented at the University of Washington. The description of the architecture is dated 1981 /E1/ and a complete architecture definition was published in 1984 /E2/. The system attempts to combine the high level of integration seen in traditional centralized systems with the advantages of distributed systems by modeling all types of resources with an active abstract data type called Eden object, or Eject. An Eject provides a number of operations that can be invoked by other Ejects, and is implemented as a group of processes. To survive crashes and to be mobile, an Eject can perform a checkpoint operation to produce a passive version that is basically a disk file. The Eject can later be reactivated using its passive representation. Ejects are location independent: they can be moved from host to host in their passive form. Using the Eject mechanism, distributed applications in Eden are implemented as a group of Ejects with an arbitrary invocation pattern. The Ejects on a host are supported by a local manager called the Eden kernel. An Eden kernel supports the processes used to implement the Ejects, provides Eject creation, migration and checkpointing operations, maintains the states of the local Ejects, and keeps a cache of the locations of remote Ejects. The collection of Ejects in an Eden system is hence partitioned into a number of groups according to their current host, and managed by the local Eden kernels. An Eject can only be accessed using a capability, and it is the responsibility of the Eden kernel to locate a remote Eject that a local user invokes.

The system was initially implemented on the Intel iAPX 432 processors (*) connected by an Ethernet. The current version is implemented on top of the Berkeley UNIX system on VAXes.

Cambridge Resource Manager

This is a distributed resource management system developed at Cambridge by Daniel Craft for the Cambridge Processor Bank /Ca1/. A logically centralized manager keeps records of processors and other types of resources in the system, and assigns them to the clients as requests come in. To make the manager general purpose, no interpretation of the resource information stored is attempted, except for magnitude comparisons. If a resource is not available, the manager consults a "cookbook" to see if it can be constructed using the available resources. All the resources are accessed through the manager, which has total control over them. This is a straightforward extension of a manager in a centralized system. To achieve higher reliability and availability, however, the resource database is fully replicated at a number of resource managers providing identical services. For every resource, one of the resource managers, called the control manager, is assigned the responsibility of maintaining loose consistency among the copies of the resource record. Since the record update frequency is reasonably low, such a full replication scheme seems to provide adequate performance.

V System

The V system is a local area network-based distributed system developed at Stanford University using a client-server model /V1, V3/. It is implemented on a collection of MC68000-based SUN workstations connected by an Ethernet. Objects are grouped into contexts that can

(*) iAPX 432 is a trademark of Intel Co.

be nested, and are supported by various types of servers. Instead of using a logically centralized nameserver to provide naming services for the objects, a distributed name interpretation scheme is used. Each server implements the naming for the objects it supports, as well as their operations, and the interpretation of an object name may be carried out by a number of servers as the components of the name are interpreted in successive nested contexts. At the beginning of this process, a per-user context prefix server maps the prefix of the user-supplied object name (a character string) into a low level server process ID, which is simply a tuple (host ID, local process ID). Then, the remaining part of the object name is sent to this server to continue the interpretation. Although some of the claimed drawbacks of a centralized name resolution scheme, such as extra overhead, low reliability, and potential inconsistency, might be avoided, it is not clear how the context prefix server gets its initial name mapping entries, and what happens if the entries become invalid. Unclear too is how the name interpretation can start if the optional context prefix is not provided by the user, or no matching entry is found in the local prefix server.

Berkeley UNIX

Berkeley UNIX 4.2 BSD is the latest release (*) of the UNIX system developed at the University of California, Berkeley /B1/. The Berkeley distributed system consists of an internet, composed of several local area networks, where each host runs the 4.2 BSD version of UNIX and is completely autonomous. A certain degree of cooperation between hosts is offered by the interprocess communication scheme. 4.2 BSD provides support for interprocess communication both within a host and between different hosts of the internet, based on the "socket" abstraction mechanism supported by the TCP,UDP/IP protocols. On top of this are supported a number of distributed applications, called well-known services, such as remote file copy (rcp), remote login (rlogin), remote shell (rsh), and remote program execution (rexec). These applications are usually implemented by having a "daemon" process running on each host for each distributed application and accepting requests from remote users on a well-known port. A local process is created by the daemon upon request to execute the task, and can be regarded as an extension to the remote logical parent's environment. Each host has a database of all the well-known services in the system. Each entry in the database contains the quadruple (service name, port number, protocol, alias). Port numbers are assigned statically to services and have to be unique (well-known ports). A number of system calls are offered to query the services database. On each host, at boot time, an initial daemon (inetd) is created, which listens for connections on certain internet sockets. When a request is received on one of its sockets, the inetd reads in a configuration file to find out what service the socket corresponds to, and invokes a program to service the request. Essentially, inetd allows one master daemon to invoke several other daemons. The system is best characterized as a network operating system because of the complete autonomy of the hosts.

Xcode system

The Xcode system is a local distributed system designed and implemented at the Politecnico di Milano within the Cnet project /C2, C3/. The basic hardware configuration of the system consists of a set of Olivetti M40s connected by an Ethernet. The software architecture is built upon these hosts and consists of two layers: a user layer, composed of a set of abstract machines, called

(*) The 4.3 BSD release is expected to be ready for distribution this Fall (1985).

Virtual Nodes (VNs), and a lower layer, which includes the run-time support for Virtual Nodes and the management of the hosts' resources. More than one Virtual Node can be supported by the same host, but a Virtual Node cannot be split among hosts, and no shared memory is provided between Virtual Nodes. Resources defined in a Virtual Node, i.e., processes and local memory (frames), are private. At the Virtual Node level, the placement of a VN to a host is unknown. (Virtual Nodes are location independent.) Each host has an Allocation Manager which is responsible for controlling and scheduling the Virtual Nodes of the host. The Xcode system can be dynamically configured, and for this purpose the Allocation Managers offer a set of "programming in the large" primitives for controlling and modifying, at run-time, the locations of the Virtual Nodes.

Unix United (Newcastle Connection)

This is an attempt to imitate a centralized system using a group of homogeneous UNIX systems by inserting a layer (the Newcastle Connection) between the user code and the system kernel /Nwl/. The hierarchical file system structure of UNIX is extended to include multiple machines as subdirectories in a global directory structure. Transparency is achieved by intercepting file operations involving files stored on remote systems and forwarding the operations to the inappropriate remote system. A "spawner" process runs constantly on each machine to accept remote file system calls and create a file server process for each remote u: pon request. The work is elegant in that the distributed system is capable of transparently supporting users on different hosts while neither the user programs nor the kernel has to be changed.

*System R**

This system, as well as the Grapevine and Clearinghouse systems to be discussed later, are not complete distributed systems as the ones we discussed above, but we decided to include them because they provide interesting approaches to resource management. R* is a distributed relational database system developed at the IBM San Jose Research Laboratory /R1, R2/. Database objects (relations, views,...) are stored in hosts that are connected by networks which may be geographical. The locations of all the objects and their access information are stored in object descriptors that collectively form the distributed database catalog. On each host, there is a local database catalog component that includes descriptors for all the objects residing on that host, as well as all the objects created there and migrated later to some other host. The users access the database exclusively through a local instance of the distributed database manager, which is responsible for locating remote objects and contacting the remote managers to get the descriptors. An object is uniquely named by a System Wide Name, which is the quadruple (user ID, user site, object name, object creation site). Since the manager at the creation site of an object is required to maintain an entry for this object in its database catalog, an object can always be located given its name. To facilitate flexible object naming, however, name completion rules and a per-user synonym table are used to map user supplied "print names" into the System Wide Names. It is interesting to note that the system avoids the use of a centralized or global catalog (nameserver) by including the location of the database manager for an object (object creation site) as part of the object's name, thus making the mapping from the object name to its manager trivial. This scheme seems to be sufficient for a distributed database with relations stored on a limited number of remote hosts and being relatively stable. However, a migrated object may become inaccessible

if its creation site goes down, even if its current host is available.

Grapevine and Clearinghouse

Both are distributed services designed and implemented at Xerox on the Xerox geographical internet /D5/. The internet is composed of a collection of Ethernet local area networks, gateways, and long distance links that connect together personal workstations and shared servers to form a company-wide or even multi-company information network. In such an environment, uniform and convenient mail, resource location, and authentication facilities are very important, and the two systems address these issues.

Grapevine is a distributed mail system on a wide-area network /X1, X2/. It also provides facilities for authentication, resource location, and access control. The objects (users, hosts, and services) are organized in a two-level hierarchy; their names have the format "*local_name.registry*", where registries represent human organizations within the environment. The most distinctive feature of Grapevine is its distributed and replicated implementation. A number of Grapevine servers maintain parts of the object space at the granularity of individual registries, and collectively support user queries in a transparent manner. This is achieved by fully replicating the server configuration data in every server, and cooperating with client interface to redirect user queries to a responsible server. The system can tolerate single server failures because the information of objects in each registry is stored in multiple servers. Also interesting is the notion of cooperation between the registration service, which maintains the naming database of the objects and supports query forwarding, authentication and mailbox location, and the message service, which delivers user mail and propagates system configuration updates for the registration service. The system is supported by a number of dedicated computers at various geographically separated locations. Although the system scale is limited by the fixed-level hierarchy, several years of operating experience show that it is capable of supporting a large organization. The Grapevine system is currently used by the Xerox Corporation, with more than 4000 users on different sites in the United States, Canada, and England.

Clearinghouse is a general service for naming and locating abstract objects in a large, distributed environment /X3/. Examples of object types supported are workstations, servers (e.g., file servers, print servers, mail server, and Clearinghouse servers), human users, and groups of these. Clearinghouse maps an object's name into a set of properties that are object type-specific, and collectively describe the object. Similar to Grapevine, Clearinghouse is decentralized and replicated, and provides transparency. To support a larger environment, however, a three-level hierarchical naming structure is used. To preserve generality of the naming service, no interpretation of the object information stored in Clearinghouse is attempted. This limits the capability of the system to a passive data repository that stores information and serves it to clients.

3.2. Taxonomy

In this subsection, we present a taxonomy of resource management schemes. The taxonomy is based on a set of attribute classes, and relies on the following assumptions:

a) Each host composing the Local Distributed System (LDS) is associated with one and only one environment containing all the objects physically located on the host.

b) No other environments are defined in the LDS other than the ones associated with the hosts.

c) All environments are disjoint.

d) Each host has associated a derived domain containing all the object names (sharable or not) that are visible from that host.

e) Any derived domain defined in the LDS contains object names whose location (i.e. placement) is in one (or more) of the hosts' environment(s) defined for the LDS.

f) We consider the problem of object management only for sharable objects (either active or passive) . In this scenario, the concept of object sharability partitions the objects of the LDS into two classes: those that are internal to a host, hence not accessible directly from other hosts (non sharable objects), and those that can be invoked and used from outside the host (sharable objects).

Classes and relative attributes are described. The classes are presented in four groups of related subjects: nature of the LDS, sharable object characteristics, object control, and user control. The results of the taxonomy are summarized in Table 1.

3.2.1. Nature of the LDS

The following classes characterize the nature of the LDS.

Distributed System

Among the possible types of LDS, we can characterize four types of systems with the following attributes: network, client/server, fully integrated, and partially integrated.

network: This attribute characterizes an LDS with the following properties. The network LDS address space is composed of a set of derived domains, one for each host . Host derived domains are not disjoint, but the common parts include a limited number of object names. That is, the number of sharable objects is small (in comparison to the number of all objects in the system). There is no sharable object replication, and, for structures composed of passive objects, or active and passive objects, the placement must be the same. Therefore, an active object can control only (passive) objects local to the host where the active object is located. The only form of interaction between different hosts is given by structures of active objects with different placements. That is, an active object, in order to control a remote passive object, has to contact a remote active object which is located on the same host as the passive object. Conditional expressions performed by active objects are based on the states of objects with the same placement as the active object. This means that object management is provided on the basis of local state information, and no kind of global state information is kept (we are not considering global names as part of the state).

The Berkeley UNIX system is a typical example of a network distributed system. Each host is an autonomous UNIX system with only a limited number of sharable objects listed on each host, in a (static) configuration file. Managers (daemons in UNIX terminology) are responsible only for objects located on the same host, and the control is based only on local state information. A number of static daemons provide a set of callable services associated with unique port numbers. A request for a service from a host other than that where the daemon is located corresponds to a request for a connection to a particular port.

client/server: This attribute identifies a large class of LDS's. For these systems, it is possible to define a set of host derived domains such that, for all the active object groups of these domains forming structures with other active object groups of other domains, their associations are always unidirectional and with orientation toward them. That is, there exists a set of hosts (servers) that with respect to a proper set of other hosts(clients), provide only callable services and not vice versa.

In this kind of systems, each host has a derived domain, but the sharable part with other domains is much larger than in the case of network systems. This corresponds to a restriction of the hosts' autonomy. Services are spread throughout the system, and completion of a job may require several interactions with different hosts. Structures of active-passive objects do not necessarily have to have the same placement, and sharable object control can be done according to some state information which is not only local. Sharable objects may be replicated, and their diffusion may vary with the "visibility" of the service they provide in the system.
Examples of these systems include the Cambridge Resource Manager and the V system. Grapevine and Clearinghouse, even though not examples of LDS, are based on the client/server model. In the Cambridge Resource Manager, the class of sharable objects is very large; it includes low level objects such as processors as well as high level objects such as compilers and other general services. Sharable objects are managed in a logically centralized manner by a Resource Manager, and are assigned for a fixed period of time exclusively to the user who requests them. In this way, sharable objects are either public available or assigned to a specific user. An assigned sharable object cannot be used by other users. Complex sharable objects can be "built" from available simpler sharable objects.

fully integrated: This kind of system is structured to appear to the user as a centralized system. Each host is functionally complete, and autonomy is extremely limited. The set of sharable objects is identical for every derived domain in the system. That is, object sharability is a system-wide property. Placement of structure is not constrained. Sharable objects may be replicated. Object control is done on the basis of some global state information. Examples are LOCUS and the Newcastle Connection . In both cases, a global file system provides a set of system-wide sharable files.

partially integrated. These systems are in between client/server and fully integrated systems. Object sharability, as in the client-server model, is relative to a subset of derived domains and corresponds to a partial distribution of the services in the system. Unlike the client-server model, however, it is not possible to define a set of derived domains (and corresponding hosts) with the properties of one-directional orientation for associations as described above. This means that each host, unlike in the client-server model, can potentially act as a server and/or client, depending on the type of services the sharable objects on the host provide, and on the applications. Examples are Eden, with Ejects, and Xcode, with Virtual Nodes.

Hosts nature

This class distinguishes between LDS's consisting of hosts with the same hardware and systems software(**homogeneous),** and LDS's composed of hosts of different hardware or software(**heterogeneous).**

3.2.2. Sharable objects characteristic

The following classes characterize the nature and the type of sharable objects.

Object Mobility

This class characterizes sharable objects according to whether or not some of them are allowed to move after they have been created, that is to change their placement.

absolute mobility: Objects may move from host to host at any time, even while performing operations (in the case of active objects) or being used (in case of passive objects).

conditional mobility: Objects must be quiescent in order to be to moved. There must not be any ongoing operations being performed on them or by them. For example, Eden and Xcode objects (Eject and Virtual Nodes, respectively) can move, but only when they are not active.

Immobility: Objects cannot move. Berkeley Unix, the V system and Cambridge are examples of systems in which objects are immobile.

Replicated objects

This class characterizes systems according to whether or not sharable objects have a multivalued placement (i.e.,location). Sharable objects may belong to more than one environment in order to increase their availability. This class distinguishes systems with and without some sort of sharable object replication by the attributes **yes** and **no.** When objects are replicated, the system has to enforce some degree of consistency among the different "copies". The subclass *consistency* is used with the following attributes: **absolute** to indicate that the system ensures consistency of all object copies after every update of a copy, that is, the object update is accepted only after a commit point has been reached; **eventual** when consistency is not enforced soon after an object update, but is guaranteed to be achieved with an unpredictable delay.

An example of a system that does not manage replicated objects is Berkeley UNIX. As a consequence, no consistency is guaranteed for any object that the user has duplicated (e.g., files). LOCUS is an example of a system which manages replicated objects (i.e.,files), with an absolute consistency, while Grapevine manages replicated objects with eventual consistency.

3.2.3. Object Control

These classes represents the ways in which objects are controlled. We consider structures composed of active and passive objects.

Structures

This class distinguishes the possible structures composed of groups of active and passive objects according to the placements of their objects:

local objects in a structure must have the same placement(i.e. location). Examples are Eden and Berkeley Unix.

remote objects in a structure may have different placements. Examples are Cambridge and LOCUS.

Placement

When a reference to an object is issued, the problem is how to find out the placement(i.e., the location) of the object on the basis of its name. Mobile objects usually make the placement more difficult, but placement problems exist also for immobile objects. This class has four non mutually exclusive attributes:

none: The placement information is provided in the name of the object. Examples are the V System and R*.

broadcast: If the placement of the destination is not known, then a broadcast query is sent to obtain the placement.

local (cache): The information is locally stored (for example cached) on each host, but without guarantee to be up to date in the case of object migration. An example is the Newcastle Connection.

specialized server: A specific active object (or a set of objects) performs the task of obtaining the object placement. All queries for object placement are performed by such an object. Examples are Grapevine and Clearinghouse.

Conditional Operations Knowledge

An active object may perform conditional expressions. This class identifies the kinds of state information that may be used in the predicates. In particular, the attributes **local** and **global** refer to whether the state information on which the predicate is based regards objects with the same placement (local) as the active object performing the conditional expression or not (global).

3.2.4. User control

The following class identifies the degree of object control available to a user (at the application level).

Transparency

location transparency: This attribute reflects the fact that the placement of a sharable object may be invisible to the user. That is, the name of the sharable object does not contain any information related to the object's location. Examples are LOCUS (for files), Xcode (for Virtual Nodes), Eden (for Ejects), and the Newcastle Connection (for files).

control transparency: The mapping (i.e., allocation) from sharable objects to hosts is hidden and/or not controllable by the user. Examples are LOCUS, and Newcastle Connection, where the user cannot modify the locations of the files on the hosts.

execution transparency: A user program can still run even if sharable objects moved (absolute or conditional). An example is Xcode, where a dynamic reconfiguration does not alter any distributed applications. Examples of systems that do not have any degree of transparency are Berkeley UNix and Cambridge Resource Manager.

Table 1. Taxonomy of Systems Surveyed

CLASS	LOCUS	Eden	Cambridge	V System	Berkeley	Xcode
Distributed system	fully integrated	partially integrated	client/ server	client/ server	network	partially integrated
Host nature	hom.	hom.	heter.	hom.	hom.	hom.
Object mobility	conditional	conditional	immobile	immobile	immobile	conditional
Replicated object	yes/absolute consistency	no	no	no	no	no
Structures	remote	local	remote	local	local	local
Placement	local	local/ broadcast	specialized server	none	local	local/ broadcast
Conditional operation	global	local	global	local	local	local
Transparency	location control execution	location execution	no	location	no	location execution

4. GLOBAL OBJECT MANAGEMENT: A PROPOSAL

In Section Two, we set up a model of object management. We used this model in our survey of a number of existing systems and in our taxonomy of object management in Section Three. In this section, we will use the results of the preceding sections to motivate our proposal of a global object management approach.

4.1. Approaches to Object Management

A local distributed system as defined in Section One is basically a group of hosts/workstations connected by a local internet. Using the terminology developed in Section Two, we consider the set of objects residing on a host as forming an environment. These objects can be classified into two types: those that are internal to the host, hence not accessible directly from other hosts, which we call nonsharable objects, and those that can be invoked and used from outside the host, which we call sharable objects. For example, a page of main memory and a physical device are managed by the local memory manager and device controller, respectively, thus may be considered nonsharable objects. On the other hand, files and print servers are examples of sharable objects. Sharable objects are usually constructed using nonsharable objects. A file, for instance, is basically an ordered list of disk blocks. Similarly, a print server is a higher level abstraction of a printing device. From the view point of object management in a distributed system, we are mainly interested in the sharable objects because the management of nonsharable objects concerns only the individual hosts, and has been studied extensively in centralized systems research. Based on this observation, we define the *global state* of an LDS to be the collection of the states of the sharable objects in the system. Object management in an LDS is then the decision making process that utilizes and manipulates the system state.

There are a number of approaches to the object management problem, as typified by the systems discussed in Section Three, but most of them can be characterized by a distributed scheme in which a number of managers are responsible for subsets of the sharable objects in the system, typically local, and provide them to the rest of the system. Consequently, the global system state is *partitioned* into disjoint parts and maintained at different locations. This is not desirable for resource sharing, because the current location of an object has to be known to the client who wants to invoke it. It is also virtually impossible in such systems to make resource allocation decisions based on system-wide resource utilizations. A number of measures have been taken to improve the situation. Local caches are employed to store information about remote objects, or the information is replicated on every host, or, if all these do not work out, broadcast is used to find the objects, thus incurring high costs. This situation can probably be best illustrated by considering some of the systems we already surveyed in the previous section.

LOCUS:

Its global file system is partitioned into file groups, each being managed by its Current Synchronization Site (CSS). All the file operations have to go through the CSS's so that synchronization can be enforced, and atomic transactions implemented. Mapping from a file name to the corresponding CSS is performed in the user host through a fully replicated table.

Newcastle:

Similar to LOCUS, a full-replication approach is used. Each host has a "mapping table" that is static and contains, for each non-local object, information for accessing it.

Eden:

A remote Eject is located in three steps: (1) a local cache is checked; if the Eject is not found, (2) a few places are checked according to the hint included in its capability, in the hope that the Eject or a forwarding pointer to it might be found; if still no luck, (3) a broadcast message is sent to find it /E3/.

Xcode:

All sharable objects are modeled by an active object type called Virtual Node, which is mobile. The locating of a remote Virtual Node is performed in two steps: (1) a local cache is checked; if not found, (2) a broadcast message is sent to find it. This resource location method of local cache plus broadcast is very similar to that used in Eden.

*R**:

The name of the creation site of an object is used as part of the object's name, making the locating of its manager trivial. If the object has moved, and its creation site is not up, however, the users may not be able to access it, even though the host containing the object is available. To avoid using the complete object name every time, some name completion rules and a local synonym table (an aliasing mechanism) are used, but they still do not solve the above problem.

V System:

Object name resolution is performed in a distributed fashion, with the location of the initial manager being part of the object's System Wide Name.

It should be pointed out that the above design decisions may be reasonable for the particular systems, but they are clearly not satisfactory for a large scale, dynamically changing distributed system such as an LDS.

The problems with this "partition" scheme can be summarized as follows:

(1) Object availability depends on the local availability of its information, thus impeding resource sharing. Embedding the name or location of an object's manager in its name or relying heavily on local caches makes object migration very difficult. Forwarding pointers may be used, but additional overhead and, even worse, additional points of failure are introduced, not to mention the garbage collection problem.

(2) If the object data is replicated, then we are faced with the classical problem of data consistency. This is aggravated by the large number of copies around (on the same order as the number of hosts in the system). This leads to the next problem.

(3) The system does not scale at all. Most of the systems above have at most several tens of hosts, but we predict that systems in the near future will be of larger scale, with many sharable objects of the same type available.

(4) Besides data replication, the object locating functionality is also replicated on every host. This has the disadvantages of being inefficient and complicating every host.

(5) Distribution of global state information makes it virtually infeasible to implement more sophisticated object selection and manipulation mechanisms in an attempt to better utilize the system resources. The loads of objects of the same type may vary greatly, some overloaded, others under-utilized.

Considering the above shortcomings of the distributed management scheme, a question naturally comes to mind: why not use a logically centralized, global agent to manage the sharable objects? In other words, why not define a well-known service in an LDS that maintains and influences the states of the sharable objects and performs object location and selection operations upon clients' requests?

Part of this idea is already present in the concept of a nameserver. A nameserver is a logically centralized binding agent that maintains object information (e.g. addresses) and performs the mapping from object name to its attributes on demand. However, since a nameserver is merely a passive data repository that serves object information without interpretation, it cannot make any decision, or initiate any operation. We propose to extend the nameserver model to a global manager that is capable of performing more complicated selection operations and of actively influencing the states of the sharable objects. More specifically, we define a *global object manager* (GOM) as a logically centralized agent in an LDS that maintains a database of sharable object information and coordinates between different parts of the system by issuing advice instead of commands. The concept of the GOM is motivated by the observation that many resource management and sharing decisions require the knowledge of global state information and hence are difficult, if not impossible, to make by individual hosts who usually view the system through a "window". By defining a centralized agent, it is easier to keep reasonably up-to-date information there and to make such decisions, because now the objects only have to be registered with the GOM to make them sharable, instead of having to spread the information around, or being polled. The GOM also represents a form of economy of scale; instead of requiring each host to be able to

perform the object location operations, it can contact the GOM to receive such service, and the GOM can perform more sophisticated operations using global knowledge in an effort to increase the availability and utilization of the system resources. For instance, when a user has a big compute job that will take a long time to run on his/her local workstation, he or she may submit a description of the job to the GOM, which selects a suitable compute server for the job, taking into consideration its type and the loads on the servers. As another example, consider the print service in an LDS. There are usually a number of print servers providing different types of services (e.g., *line, letter, laser, graphics, multi-color*), and located at different places. A user may submit a print job to a printer and then regret the decision because that printer is heavily loaded. Instead, the user may specify the job requirements, like quality, location, delay, and the GOM may monitor the availability and loads of the print servers and help the users to make intelligent decisions.

It should be pointed out that a global management scheme becomes interesting only for reasonably large systems. For an environment consisting of a number of workstations, one file server and one print server, for example, the object management problem is trivial, since there is not much choice as to which object to use.

4.2. Considerations on the Design of a GOM

Several obvious problems have to be resolved in order for the GOM approach to be applicable to LDS's. First, the existence of a GOM threatens the autonomy of the individual objects and hosts. The GOM may potentially control the operation of the objects being managed, hence *sacrificing local autonomy in a distributed system. Second,* a GOM may become a central point of failure, and, since it plays such an important role in the system, such a failure would seriously impair the normal system operation. Related to this problem, a GOM is also a potential performance bottleneck. The last problem is that it is impossible for a GOM to maintain an exact copy of the distributed system's state, simply because the state is changing all the time and the GOM is physically separated from the objects. Although the four problems above are very serious, we believe that they can be avoided or alleviated. Below, we discuss a number of considerations on the design of a GOM that help solve these problems.

4.2.1. Object types

Since the GOM performs global management in an LDS, only the top-level objects should be considered, and a loose hierarchy of managers should be used. For example, although a file is a sharable object, it is not suitable for management by the GOM. Instead, we use file servers to manage the files, and use the GOM to manage the file servers. Different types of objects exhibit different needs for their management. While it seems desirable to use the GOM to schedule jobs for the large, dedicated compute servers, synchronization of the hosts' clocks should probably be performed in a distributed manner, without the participation of the GOM. In other words, it is unadvisable to put the management of every type of sharable object into the GOM, making it a monster that tries to do everything, but does nothing well. Some types of objects suitable for management by the GOM are file servers, compute servers, print servers, and user environments, which include users' mailboxes, personal information, and their global working environment.

4.2.2. Autonomy

The concept of a manager in a distributed system should be different from that in a centralized system. Since a centralized system, for instance, a single host, usually belongs to one organization, and is physically in one location, its managers are usually *controllers* for various types of objects. A process scheduler, for instance, manages the processes by allocating the processor to them. Not executing a decision made by the scheduler is considered a failure in a process or the processor. In contrast, since the objects in a distributed system often belong to separate groups or organizations, their interests may conflict, and hence local autonomy is desirable. A manager in this environment should act as a *coordinator* or mediator that tries to balance the needs of different parts of the system and to fully utilize the available resources, while honoring the autonomy of the individual parts. In particular, an object to be shared by clients on other hosts can be registered with the GOM by sending a "registration request" that includes a description of the object, such as its type, location, and capacity. Access control information may also be specified to restrict the availability of the object only to the intended clients. The GOM does not have the power to subject any object to its management. Only privacy, not autonomy, is potentially lost, because the object can still decide whether to accept an invocation. The advice given by the GOM is usually, but not necessarily, followed by the clients and by the objects. This is in contrast, for example, to the Cambridge Resource Manager, which has total control over the pool of processors and other types of resources.

4.2.3. Performance and Reliability

Regarding the performance and failure problems, we point out that the logical centralization of the GOM does not imply physical centralization. To achieve high reliability and availability, as well as reasonable performance, the GOM has to be implemented as a replicated system. There is a big difference between replication in the GOM and that in every host, because the degrees of the two replication methods usually differ by one or more orders of magnitude. Another way to alleviate the performance problem is to restrict the types of objects to be managed by the GOM, as already discussed earlier. In Section One, we pointed out that the scale of LDS's is limited because of the locality of communication and sharing. This limitation is important for a centralized scheme to deliver reasonable performance.

In principle, every request for an object managed by the GOM could be sent to the GOM, but this would entail terrible performance. Take the file name mapping as an example again. Suppose a user wants to open a file and the local host determines that the file is not stored locally. It may send a request "**GetFileServer**(FileName)" to the GOM to find out the name and location of the file server managing this file and then send the open request to it. In addition, it may cache the mapping from the file name to the file server so that next time the same file or a file in the same directory is to be opened, it does not have to ask the GOM. In case the file server's state changed, say its host crashed, or the server moved, the cache entry becomes invalid, and the GOM has to be contacted again to find an alternative server if the file is replicated, or the new location of the same server. Past experiences in systems design have shown repeatedly that caching can improve performance significantly.

4.2.4. System State

In principle, the object states are changing all the time, and therefore it is impossible for the GOM to maintain the state of the system. In practice, however, this problem can be eased by two factors. First, the GOM does not have to maintain the full version of the system state, but just the object information that will enable it to perform its management functions. For instance, the GOM does not need to know all the files being managed by a file server, but just the part or parts of the file space it manages. Second, it is not necessary for the GOM to keep an absolutely accurate copy of the system state. Some objects change their states slowly or only very infrequently. A file server is not likely to migrate now and then, and its host may not crash for days or even weeks. Even for those objects with quickly changing attributes, like the load of a compute server, an averaging operation over a period of time may be adequate to enable the GOM to make good decisions most of the time. In case the decision is wrong, the object may refuse the request from the user. For example, if a user is given a highly loaded compute server, the server may reject the user's compute job, and the user will try to find another compute server, probably by contacting the GOM later when it gets new load information from the compute servers. In other words, the GOM only provides *hints*, and it is the responsibility of the user to recover from errors.

4.3. GOM Organization

The structure of a GOM should meet the requirements discussed in the previous section. We study GOM organization along two lines: logical and physical.

4.3.1. Logical Structure

The logical structure, or the architecture, of a GOM should facilitate modular growth of the system. Specifically, the management algorithms of a particular object type should be easily modifiable without affecting other types. Moreover, as the system evolves, needs for managing new object types using the GOM may emerge, and should be accommodated gracefully. These requirements dictate that the architecture should be modular and in layers. The bottom layer of the GOM provides the functionalities of a nameserver, that is, an object database and a set of basic queries. The GOM is accessed through a common interface that routes a request to the object type-specific management module based on the type of the request. All the knowledge about a particular type of object and the corresponding algorithms are localized in the manager module, and hence are easily changeable. The manager modules access the object database through the well-defined internal nameserver interface. This structure also makes adding new object types easy; only a new management module needs to be added that uses the existing interfaces to the other parts of the GOM, while no part of the system has to be changed except that the new type has to be "announced" to the GOM interface.

4.3.2. Physical Structure

To make GOM service highly reliable and available, replication techniques have to be used. Classical studies of this problem provide us with the choices of multiple-copy redundancy, where several physical instances of the GOM maintain (almost) identical object database and provide identical services, the primary-secondary scheme, where updates are only accepted by the primary

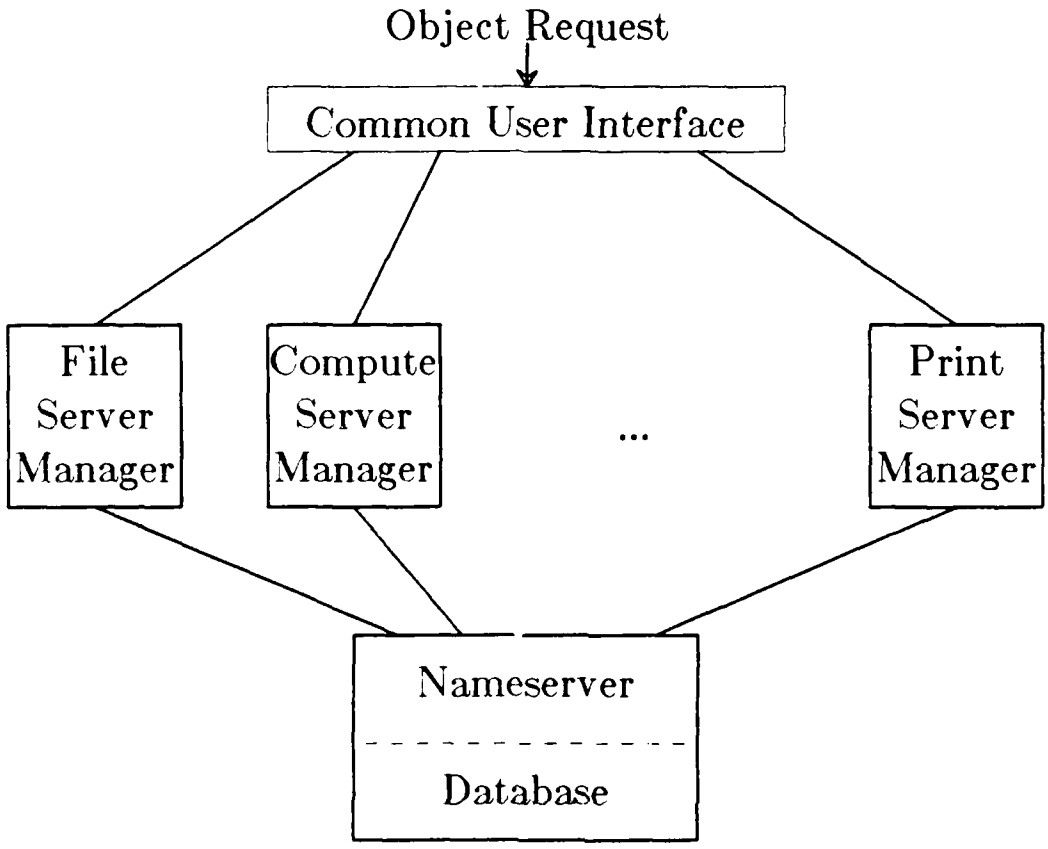


Figure 4.1 Logical structure of the GOM

GOM, and the active-standby model, where the functionality is physically centralized in the primary GOM that propagates the updates to the standby GOM in preparation of a failure. Increased performance and reliability have to be balanced against the increased amount of overhead spent maintaining consistency among several databases. The impiementation choice has to be made on a system-by-system basis.

5. CONCLUSIONS

Resource management in distributed systems is both an interesting and a difficult problem. The research presented in this paper was prompted by our dissatisfaction with the existing approaches, and the recognition of the importance of the problem in distributed systems research. We identified the type of system we are interested in by defining Local Distributed Systems and their characteristics. A model of resource management was established and used in an overview of the approaches taken in a number of distributed systems. Based on that study, we proposed a new approach to the problem, the global object management. We feel that this approach offers a number of potential advantages over the existing ones, and may lead to a distributed systems design methodology that is conducive to extensive resource sharing. Potential problems with the approach were discussed, and possible solutions given. The structure of a Global Object Manager

was also briefly studied.

It should be pointed out that the results of this paper are qualitative and untested by system implementation experiences. Nonetheless, we feel that the idea presented pointed out a promising direction of research into the resource management problems. We plan to continue the work started with this paper. Modeling techniques will be used to verify the advantages postulated and explore appropriate structures for the GOM. More importantly, perhaps, we plan to implement an experimental system incorporating the global management approach. It is our hope that a combination of these techniques will provide us with quantitative arguments for global management, and reveal new areas of research.

6. ACKNOWLEDGEMENTS

Professor Domenico Ferrari read drafts of this paper and provided valuable suggestions. Joel Emer of Digital Equipment Corporation also offered helpful comments.

7. BIBLIOGRAPHY

Data Abstractions:

/A1/ Goguen, J., J. Thatcher, E. Wagner, and J. Wright, "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," Data Structuring (Current Trends in Programming Methodology), Vol. 4, R. Yeh, Editor, Englewood Cliffs, NJ, Prentice Hall, 1978.

/A2/ Paolini, P. and R. Zicari, "Properties of Views and their Implementation," Advances in Database Theory, Vol. 2, Jack Minker et al. editors, Plenum Press, New York, 1984.

/A3/ Gottlob, G., P. Paolini, and R. Zicari, "Properties and Update Semantics of Consistent Views," August 1985, submitted for publication.

/A4/ Claybrook, B., "Defining Database Views as Data Abstractions," IEEE-TSE, SE-11, 1, Jan. 1985.

Berkeley UNIX:

/B1/ Joy, W., E. Cooper, R. Fabry, S. Leffler, K. McKusick, and D. Mosher, "4.2BSD System Manual," Computer Systems Research Group, University of California, Berkeley, July 1983.

Xcode (Cnet):

/C1/ Lijtmaer, N., "The First One Hundred Cnet Abstracts," Cnet Report, No. 101, IEI-CNR, Pisa, November 1983.

/C2/ Tisato, F., and R. Zicari, "The Xcode Machine," Proc. of the Third Symposium on Microcomputer and Microprocessor Applications, Budapest, Hungary, October 1983, and ACM Sigsmall Newsletter, 10, 1, 1984.

/C3/ Tisato, F., and R. Zicari, "Mechanisms for Dynamic Configuration of a Locally Distributed System," IEEE 10th Conference on Local Computer Networks, October, 1985, Minneapolis.

/C4/ Ronzani, S., F. Tisato, and R. Zicari, "An Office Specification Language Based on Path Expressions," OFFICE AUTOMATION 85 (ACM), October 2-4, 1985, Erlagen, West Germany

/C5/ Zicari, R. "On Dynamic Configuration of Distributed Programs," Proc. 6th Congress on Information Processing and Automation Control, Oct. 15-18,1985, Madrid, Spain.

/C6/ Zicari, R., "Reliability and Dynamic Configuration of a Distributed System," Proc. 5th IEEE Symposium on Reliability in Distributed Software and Database Systems, Jan. 13-16,1986, Los Angeles, CA.

Cambridge Distributed System:

/Ca1/Craft, D., "Resource Management in a Decentralized System," Proc. of the Ninth Symposium on Operating Systems Principles, ACM, Bretton Woods, NH, Oct. 1983, pp. 11-19.

/Ca2/Richardson, M. and R. Needham, "The Triplos Filing Machine, a Front End to a File Server," Proc. of the Ninth Symposium on Operating Systems Principles, ACM, Bretton Woods, NH, Oct. 1983.

CMU:

/Cm1/Daniels, D. and Z. Spector, "An Algorithm for Replicated Directories," CMU Tech. Rep. CS-83-123, May 1983.

/Cm2/Jones, M., et al, "Matchmaker: an Interface Specification Language for Distributed Processing," CMU Tech. Rep. CS-84-161, December 1984.

general:

/D1/ Abraham, S. and Y. Dalal, "Techniques for decentralized management of distributed systems," Proc. 20th COMPCON, San Francisco, CA, February 1980.

/D2/ Clark, D. and L. Svobodova, "Design of distributed systems supporting local autonomy," Proc. 20th IEEE COMPCON, February 1980.

/D3/ Enslow, P., "What is a 'distributed' data processing system?" IEEE Computer, 11, 1, January 1978.

/D4/ Gottlob, G. and R. Zicari, "Properties of a Multilevel Hierarchical System Structured with Data Abstractions," *in preparation.*

/D5/ Jensen, D., "Decentralized Executive Control of Computers," Proc. of 3rd ICDCS, pp. 31-36.

/D6/ Lampson, B.,M. Paul, and H. Siegert, editors, "Distributed Systems---Architecture and Implementation," Springer Verlag, 1981.

/D7/ Lantz, K., K. Gradischnig, J. Feldman, and R. Rashid, "Rochester's Intelligent Gateway," IEEE Computer, Oct 1982.

/D8/ Metcalfe, R. and D. Boggs, "Ethernet: Distributed packet switching for local computer networks," Comm. ACM, 19, 7, July 1976.

/D9/ Olivetti Co., "M40 Architecture," Olivetti Report, 1982.

/D10/Rashid, R. and G. Robertson, "Accent: a Communication Oriented Network Operating Kernel," Proc. of the Eighth Symposium on Operating Systems Principles, ACM, Pacific Grove, CA, May 1981, pp. 64-75.

/D11/Saltzer, J., "Research Problems of Decentralized Systems with Largely Autonomous Nodes," Operating System Review, 12, 1, January 1978.

/D12/Stankovic, J., "A perspective on distributed computer systems," IEEE Transaction on Computer, C-33(12), December 1984.

/D13/Svobodova, L., "Workshop Summary-Operating Systems in Computer Networks," Operating Systems Review, 19, 2, April, 1985.

/D14/Terry, D., "An Analysis of Naming Conventions for Distributed Computer Systems," Proc. ACM SIGCOMM 84, Montreal, Quebec, June 1984.

/D15/Zhou, S., "Design and implementation of the Berkeley Internet Name Domain (BIND) Server", Technical Report, University of California, Berkeley, Computer Science Division, No. 177, Fall, 1984.

Eden:

/E1/ Lazowska, E., H. Levy, G. Almes, M. Fisher, R. Fowler, and S. Vestal, "The Architecture of the Eden System," *Proc. of the Eighth Symposium on Operating Systems Principles*, ACM, Pacific Grove, CA, May 1981, pp. 148-159.

/E2/ Almes, G., A. Black, E. Lazowska, and J. Noe, "The Eden System: A Technical Review," IEEE Transactions on Software Engineering SE-11, 1, January, 1985, pp. 43-59.

/E3/ Black, A., "Supporting Distributed Applications: Experience with Eden," Tech. Report 85-03-02, University of Washington, March 1985

Locus:

/L1/ Popek, G., B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudsin, and G. Thiel, "LOCUS: A Network Transparent, High Reliability Distributed System," Proc. of the Eighth Symposium on Operating Systems Principles, ACM, Pacific Grove, CA, May 1981, pp. 169-177.

/L2/ Parker, D. et al., "Detection of Mutual Inconsistency in Distributed Systems," IEEE Transaction on Software Engineering, May 1983.

/L3/ Walker, B., G. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS Distributed Operating System," Proc. of the Ninth Symposium on Operating Systems Principles, ACM, Bretton Woods, NH, Oct. 1983, pp. 49-70.

/L4/ Mueller, E., J. Moore, and G. Popek, "A Nested Transaction Mechanism for Locus," Proc. of the Ninth Sysmposium on Operating Systems Principles, ACM, Bretton Woods, NH, Oct. 1983.

/L5/ Page, T., and G. Popek, "Distributed Data Management in Local Area Networks," Proc. PODS, Portland, Ore, 1985.

/L6/ Page, T., M. Weinstein, and G. Popek, "Genesis: A Distributed Database Operating System," Proc. ACM SIGMOD, 1985.

/L7/ Weinstein, J., et al, "Transactions and Synchronization in a Distributed Operating System," University of California, Los Angeles, 1985.

Newcastle:

/Nw1/Brownbridge, D., L. Marshall, and B. Randell, "The Newcastle Connection, or UNIXes of the World United!" Software-Practice and Experience, 12, 1982, pp. 1147-1162.

R*:

/R1/ Lindsay, B., L. Haas, P. Wilms, and R. Yost, "Computation and Communication in R*: A Distributed Database Manager," ACM Trans. on Computer Systems, 2, 1, Feb. 1984, pp. 24-38.

/R2/ Lindsay, B., "Object Naming and Catalog Management for Distributed Database Manager," Proc. Second International Conference on Distributed Computing Systems, Paris, France, April 1981, pp. 31-40.

Xerox:

/X1/ Birrel, A., R. Levin, R. Needham, and M. Schroeder, "Grapevine: An exercise in distributed computing," Comm. ACM, 25, 4, April 1982, pp. 260-274.

/X2/ Schroeder, M., et al, "Experience with Grapevine: The growth of a distributed system," ACM Transactions on Computers, 2, 1, February 1984.

/X3/ Oppel, D. and Y. Dalal, "The Clearinghouse: a Decentralized Agent for Locating Named Objects in a Distributed Environment," XEROX, OPD-T8103, October 1981, and ACM Trans. on Office Information Systems, 1, 3, July 1983, pp. 230-253.

V System:

/V1/ Cheriton, D. and T. Mann, "Uniform Access to Distributed Name Interpretation in the V System," Computer Science Department, Stanford University, 1984.

/V2/ Cheriton, D. and Zwaenepoewl, "The Distributed V kernel and its Performance for Diskless Workstations," ACM TOCS, 1983.

/V3/ Lantz, K. and J. Edighoffer, "Towards a Universal Directory Service," Stanford University, October 1983.